DOI: 10.1111/cgf.142648
EUROGRAPHICS 2021 / N. Mitra and I. Viola
(Guest Editors)

Volume 40 (2021), Number 2

SnakeBinning: Efficient Temporally Coherent Triangle Packing for Shading Streaming

J. Hladky [1] H. P. Seidel [1] M. Steinberger [2]

[1]Max-Planck-Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
[2]Graz University of Technology, Austria

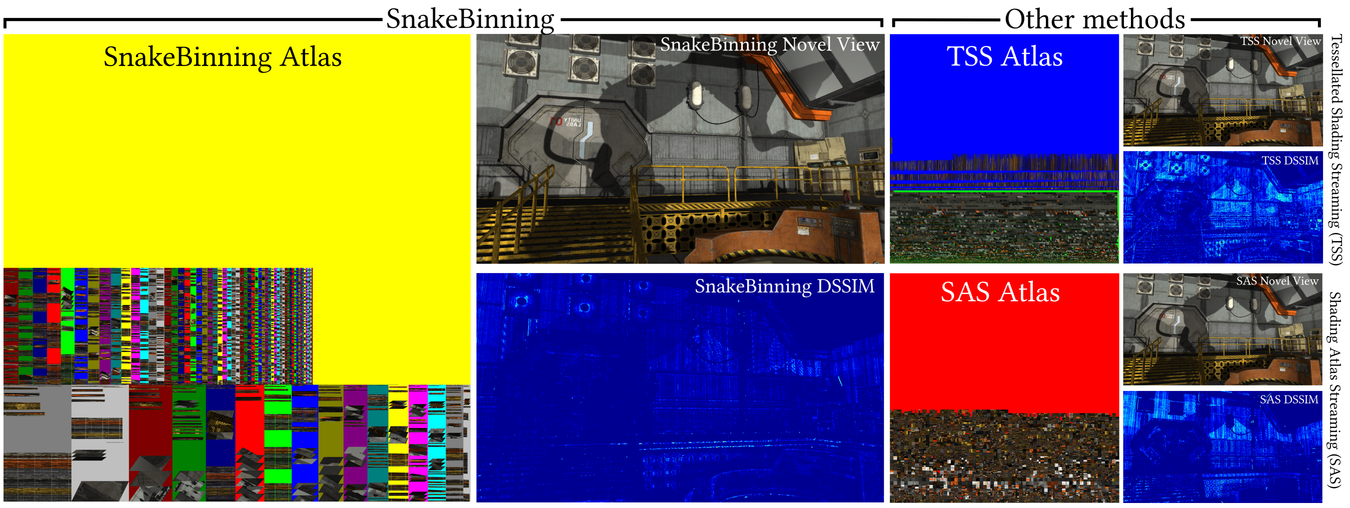


Figure 1: *By rendering Potentially Visible Set (PVS) geometry shaded with our Shading Atlas (yellow background) we achieve novel view extrapolation with near ground-truth quality. The scene triangles are assigned into bins, grouping triangles by similar screen-space footprint and preserving the primitive ordering. The bins are structured into rectangular superblocks (colorful columns) to efficiently utilize the atlas space. The gaps in the superblocks allow additional space for the bins to shrink/grow in order to increase temporal coherence. Both Tessellated Shading Streaming (TSS) [HSS19b] and Shading Atlas Streaming (SAS) [MVD*18] capture similar amount of shading atlas samples at slower speeds and achieve worse novel view quality (last column).*

Abstract
Streaming rendering, e.g., rendering in the cloud and streaming via a mobile connection, suffers from increased latency and unreliable connections. High quality framerate upsampling can hide these issues, especially when capturing shading into an atlas and transmitting it alongside geometric information. The captured shading information must consider triangle footprints and temporal stability to ensure efficient video encoding. Previous approaches only consider either temporal stability or sample distributions, but none focuses on both. With SnakeBinning, we present an efficient triangle packing approach that adjusts sample distributions and caters for temporal coherence. Using a multi-dimensional binning approach, we enforce tight packing among triangles while creating optimal sample distributions. Our binning is built on top of hardware supported real-time rendering where bins are mapped to individual pixels in a virtual framebuffer. Fragment shader interlock and atomic operations enforce global ordering of triangles within each bin, and thus temporal coherence according to the primitive order is achieved. Resampling the bin distribution guarantees high occupancy among all bins and a dense atlas packing. Shading samples are directly captured into the atlas using a rasterization pass, adjusting samples for perspective effects and creating a tight packing. Comparison to previous atlas packing approaches shows that our approach is faster than previous work and achieves the best sample distributions while maintaining temporal coherence. In this way, SnakeBinning achieves the highest rendering quality under equal atlas memory requirements. At the same time, its temporal coherence ensures that we require equal or less bandwidth than previous state-of-the-art. As SnakeBinning outperforms previous approach in all relevant aspects, it is the preferred choice for texture-based streaming rendering.

Keywords: texture-space shading, object space shading, shading atlas, streaming, temporal coherence, virtual reality

CCS Concepts
• **Computing methodologies** → **Rendering; Texturing; Virtual reality;** Image-based rendering;

1. Introduction

High-quality, real-time rendering was synonymous with powerful desktop computers in the past. However, with the rise of multiple reinforcing factors, the landscape of high quality, real-time rendering systems has seen a significant shift over the last years. High bandwidth, low latency networks and demand driven computing in the cloud shift execution increasingly to remote locations. Light-weight devices, such as mobile phones, hand-held gaming devices and inexpensive laptops, as well as smart television sets, are omnipresent and allow for high mobility. Finally, the interest in virtual reality (VR) and head mounted displays (HMD) has seen a recent surge with many companies investing in the technology. Combining these developments, it is not surprising that real-time rendering and gaming is moving towards the cloud and dedicated servers, while display and input happens on inexpensive light-weight devices.

The main issue with such streaming rendering solutions is round trip delay. An input, such as a game control change or a head movement happening on the light-weight client must be sent to server, a new frame generated and sent back to the client for display. Especially when using HMDs or when playing highly reactive games, a low round trip latency is key to generate an overall pleasant experience, and in the case of HMDs to avoid VR sickness. Current commercial game streaming systems, simply run a game instance on the server, capture the output images and transmit an encoded video stream to the client. To counteract the potentially long round trip latency, game streaming currently employs edge computing, providing many game server centers, as close as possible to the clients. Such an approach obviously increases cost and does not allow for on-demand load balancing—one of the major selling points for cloud computing. However, even when the server and client are in close proximity, such as a local desktop server and an HMD connected via WiFi, latency may already be noticeable and a short WiFi interference may lead to frame drops and VR sickness.

To counteract the latency issue, the client must be able to perform framerate upsampling and frame extrapolation—a technique even present for tethered HMDs in the form of asynchronous time warping (ATW) [Ocu18], where the rendered frame is adjusted for the current head movement right before display using a homography. While ATW is simple and can trivially be added after rendering with an extended field of view, severe artifacts become visible if adjustments are not limited to tiny offsets—as parallax and disocclusion artifacts become immediately noticeable.

As an alternative, the rendering pipeline can be split and decoupled between server and client, as seen in Figure 2.

The server determines all primitives which may become visible under maximum head/camera movement on the client, computes shading for these potentially visible primitives, encodes both the shading and geometry data and sends it to the client. The client receives and decodes this package containing the geometry and shading data. The client then renders the received geometry under the current head/camera movement, shading it with the received shading data. The client framerate can potentially be significantly higher than the rate at which server updates are received.

As shading information must include primitives that may only become visible under movement—and thus are occluded at the

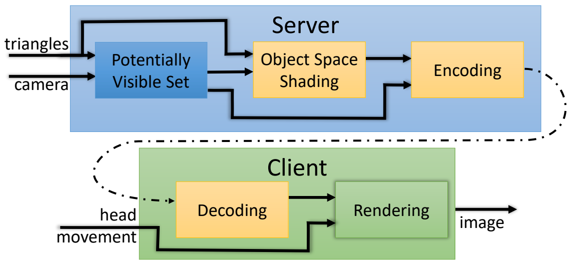


Figure 2: *Our streaming rendering pipeline. The server identifies all geometry potentially visible from any view within a supported range of camera movements and computes shading information for this geometry. Geometry and shading information are encoded into a data package and sent to the client, which then uses forward rendering to display novel views.*

current server view—the final rendered image on the server is in no suitable format for storing and transmitting shading information. As an alternative, shading can be stored in a texture atlas [MVD*18, HSS19b] and transmitted as an atlas video stream.

Previous approaches towards packing shading information into an atlas serve either of two goals. Shading Atlas Streaming (SAS) [MVD*18] maps patches of one to three triangles into rectangular blocks and allocates best fitting rectangles from the texture atlas. While this allows to keep blocks at the same location in the atlas for multiple frames, sample distributions are typically sub-optimal, as they do not consider perspective effects and compromises between triangles combined into patches need to be made. Alternatively, every potentially visible triangle can be mapped separately into the shading atlas using Tessellated Shading Streaming (TSS) [HSS19b]. By employing tessellation and special treatment of slanted triangles, samples are distributed according to screen-space shape and perspective effects. However, the special treatment for slanted triangles becomes prohibitively slow. Even more critical is that triangle locations in the atlas change every frame and thus temporal coherency is completely ignored. Thus, although samples are employed more effectively in TSS than in SAS, both approaches achieve similar quality when allotted with similar bandwidth.

With SnakeBinning, we address the shortcomings of previous atlas packing strategies and make the following contributions:

- We provide a unified rendering approach for triangles of all shapes and distortions into a shading atlas, which is efficient independent of the triangle size and slantedness.
- We propose a three dimensional binning approach to characterize triangles according to their shape on screen and organize them in a texture atlas with little wasted space.
- We establish a complete per-bin ordering of triangles across frames by exploiting the primitive order established through hardware-supported real-time rendering and thus create frame-to-frame coherency in the atlas effectively reducing the required bandwidth of the atlas stream.

We test our approach on various test scenes for streaming rendering, indicating that SnakeBinning outperforms both SAS and TSS in rendering speed. At the same time, we reduce wasted atlas space in comparison to TSS while creating more optimal sample distributions than both SAS and TSS and thus higher image quality. Finally, by relying on primitive order to sort triangles within bins and locking bin locations under small camera offsets, we achieve very high temporal coherency and thus also reduce bandwidth requirements compared to SAS and TSS.

2. Related work

SnakeBinning touches the areas of texture-space shading, image-based rendering (IBR), and remote rendering, with a special focus on atlas-based shading transmission. Commonly, these techniques explore data representations that strive to enable efficient extrapolation of high quality novel views.

2.1. Texture-space shading

Texture-space shading, often also referred to as object-space lighting or texel shading, stores shading into a texture, rather than on screen [Bak16]. It allows to exploit temporal and spatial coherency [RKLC*11] which is not possible with traditional IBR methods. Multiple variants of this kind of shading approaches have been proposed as GPU extensions [BFM10, CTM13, CTH*14, AHTAM14], which have not been realized in practice yet. On current hardware, relying on unique UV-mappings shading can be dynamically baked in texture-space [Bak16]. Similarly, Texel Shading [HY16] gathers shading into pre-charted mip-mapped textures only for visible scene portions. Unfortunately, this is not suitable for streaming as large portions of the textures remain empty.

While relying on designer created UV-layouts may prove difficult to ensure consistent quality across objects, alternative texture layouts, such as Ptex [BL08] or Mesh Color Textures [Yuk17], form a sensible alternative, as they generate samples on a per triangle bases. This approach has been picked up by TSS [HSS19b] for non-slanted triangles to effectively pack shading samples. Similar in spirit, our approach dynamically chooses a sample layout based on triangles shape and distortion while focusing on each triangle's screen projection for efficient packing.

2.2. Image-based rendering

Image-based rendering has been the go-to for various novel-view creating approaches, including the previously mentioned asynchronous time warping (ATW) [Ocu18]. As ATW only distorts the last rendered image, disocclusion cannot be revealed and reprojection needs to find a compromise between the different depths in the scene, leading to most noticeable artifacts for scenes with large depth discontinuities. Advanced IBR methods may consider the depth or distortion of individual image regions, by viewing samples as individual 3D points [CW93], using a grid [DER*10], a layered depth images [SGHS98] or a complete unstructured lumigraphs [BBM*01]. Alternatively, the depth buffer can be used for warping, creating a simple geometric proxy [MMB97]. Variants of this approach include the use of an adaptive grid [DRE*10], bidirectional search [YTS*11] and iterative search [BMS*12]. Obviously all these approaches are limited to objects visible in the previously rendered frame and the available resolution of the image and depth buffer.

2.3. Remote rendering

The more advanced warping or upsampling technique employed by a remote rendering system, the more data they also need to transmit [NCO03, SH15]. At simplest approach obviously only transmits a video stream and potentially uses ATW. Using color+depth allows for advanced warping techniques [PHE*11, CG02, SNC12]. As the depth buffer only depends on the rendered image resolution, those IBR techniques are independent of the scene's geometric complexity. Alternatively, complete frames can be rendered speculatively [LCC*15] and residual images can be transmitted [YN00, BG04, CWC*15].

For static scenes, many optimizations are possible, such as using view-dependent texture maps [COMF99], geometry images [SMSW11], or impostors [TL01, BCC16]. Similarly, for static or rigid objects, a geometric proxy can be used for perspective texture mapping [RKR*16]. Keeping a simplified or full model data on the client allows to reduce transmission to images data only [Lev95, MCO97, CLM*15].

The most related work to ours are split rendering approaches. Kahawai [CWC*15] tries to temporally or spatially augment shading on a client, which requires a rather complete client and a complicated infrastructure. To keep the client simple, Shading Atlas Streaming (SAS) [MVD*18] and Tessellated Shading Streaming (TSS) [HSS19b, HSS19a] perform shading completely on the server and pack shading into a transmitted atlas. The client renders novel views relying on straightforward texture mapping from a potentially visible set (PVS) of triangles, which is also transmitted by the server. An ideal atlas packing often requires long pre-processing times [LPRM02]. SAS follows the virtual texture packing of rectangular shapes established in Far Cry [Che15], while updating the packing dynamically. To this end, it manages empty blocks of different sizes in an atlas without block reuse. As the atlas runs full, the complete atlas memory management is reset and starts anew. In the time between atlas resets, the atlas video stream is temporally coherent. At an atlas reset the coherency is destroyed, leading to spikes in the video bandwidth. TSS addresses the shortcoming of SAS's rectangular blocks by individually determining and assigning samples to triangles. In this way, perspective effects and exact projected triangle sizes can be considered. However, TSS is not able to handle slanted triangles efficiently and does not create temporal coherency in the atlas stream, treating every frame independent of the last.

Our approach comes with the advantages of both SAS and TSS and further improves on their respective strengths and performance. Our SnakeBinning enables ideal sample distributions including perspective adjustments. It achieves a tighter packing for slanted triangle bins than TSS while being fully temporally coherent. While SnakeBinning reduces temporal coherency slightly from frame-to-frame compared to SAS between atlas resets, our approach seldom requires global resets and thus achieved better temporal coherency in the long run.

3. SnakeBinning

In order to achieve high quality novel view generation on the client, the distribution of shading samples in the atlas should closely follow

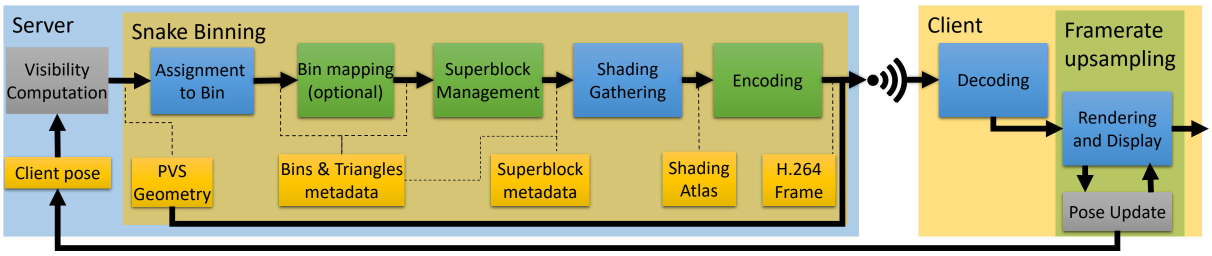


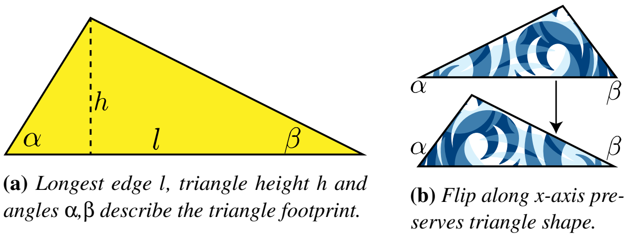
Figure 3: *SnakeBinning integrated into a streaming rendering pipeline. Starting with a PVS of potentially visible triangles, we bin triangles according to their predicted on screen footprint, optimize the bin setup, dynamically manage memory as superblocks, gather shading samples and encode the atlas stream for transmission. The client receives the geometry information and the video stream of shading information to perform framerate upsampling. Blue blocks map traditional rendering stages and green the compute jobs.*

sample distribution on the client screen. This is challenging as the projected triangles constantly change their screenspace footprint. The sample distribution needs to account for perspective foreshortening and enable efficient sample interpolation. Furthermore, the triangles should be tightly packed into the atlas to minimize wasted space as well as reduce the bandwidth load and increases the efficiency of video encoding and decoding of the atlas frames.

To map triangles into an atlas for streaming rendering, we propose a five stage pipeline outlined in Figure 3. We use three parameters to describe the screenspace footprint of a triangle and use them to group triangles of similar shape together into bins. We group bins into superblocks—rectangular portions of the shading atlas. As a first step, the server determines triangles, which may become visible in the near future, a traditional potentially visible set (PVS) problem on triangles. To this end, one can use view prediction and sampling [MVD*18] or establish visibility in an alternative space [HSS19a]. Our algorithm starts with the PVS geometry. First, in the *Bin Assignment* stage, we use the predicted next client view to assign the triangles to bins according to their screen-space footprint. Binning ensures that triangles with similar footprint are combined. Using a large number of bins ensures a tight packing, however the number of partially filled bins may be large. To alleviate this issue, the bin assignment step is periodically followed by a *Bin Mapping* step, where we map scarcely populated bins to the nearest densely populated bin. Afterwards, the bin and triangle metadata is passed to the *Superblock Management* stage, which manages the bin locations in the atlas by allocating and deallocating superblocks, *i.e.*, rectangles of a fixed height and variable width. Finally, *Shading Gathering* creates shading samples for all triangles in the PVS and stores them in the atlas, before *Encoding* the atlas frame for transmission.

3.1. Triangle Footprint

Ideally, every triangle should be sampled exactly the way it will be sampled on the client display during rendering. Since we do not know the interactions of the user within the near future, we use prediction to get an idea about which potential views the user will most likely take. To this end, we use a standard Kalman filtered motion prediction [KEP97] and sample prediction frames uniformly spaced in time. Among the prediction frames, we determine the largest footprint of each triangle, producing the densest sample



(a) *Longest edge l, triangle height h and angles α, β describe the triangle footprint.* **(b)** *Flip along x-axis preserves triangle shape.*

Figure 4: *To bin triangles, we use the height h on top of the longest edge l and the angles α and β. Flipping triangles along the x-axis ensures that all triangles follow the same shape and $\alpha \geq \beta$.*

distribution. Without loss of generality, we simply denote the view which leads to highest density/number of samples the reference view. For triangles that are not clipped or culled it is typically the view closest to the triangle.

Given the largest triangle footprint, we calculate its projection to the client screen to determine how many samples a triangle should receive. The projection varies, depending on where on screen a triangle is—it could even be partially culled or reach behind the camera. To this end, we set up a *virtual camera* for each triangle by rotating the camera such that the triangle is centered and measure its size, *i.e.*, determine all edge lengths in pixel on the virtual client view. In some cases the triangle may still reach outside of the screen or behind the camera—for these cases, we compute a new camera position that encompasses the whole triangle in its frustum, by moving the camera "backwards"—*i.e.* in the reverse lookat direction. According to our tests, while mirroring a spherical projection to determine the required sample count [MVD*18] improves the measurement of triangles intersecting the frustum, it also tends to oversample all triangles entirely contained within the frustum. Moving the virtual camera "backwards" to encompass the whole triangle corrects frustum-intersecting triangles while handling the in-frustum triangles correctly. This strategy performs well for most cases (99.99% of geometry in our tests). However, for the extreme case when a single triangle covers almost the entire screen after projection and continues to protrude out of the viewing frustum, this approach leads to noticeable undersampling. We identify such cases during the *Bin Assignment* stage and slice these triangles using frustum clipping, thus introducing a small number of new geometry.

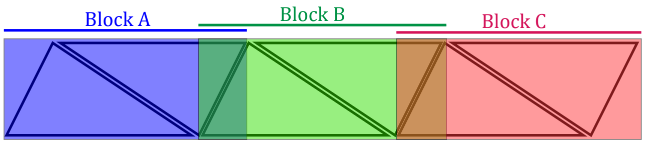


Figure 5: *A 6-triangle strip, split into 3 blocks - A, B, C. Note that blocks overlap.*

In our tests the pixel coverage was seldom dominated by a handful of triangles ($< 0.01\%$), therefore we can afford such special treatment for this corner case.

3.2. 3D Binning

Ideally, we would want to render the triangle exactly as determined from the footprint, staying true to the triangle edge lengths, area, angles, and rotation on screen. When rendering the triangle to the atlas, we also need to consider perspective effects on the sample distributions to generate shading akin to how the triangle will most likely be viewed. However, packing arbitrarily shaped triangles into an atlas is a strongly-NP hard problem, when considering its relation to the strip packing problem [BCR80]. To find a solution to the packing problem that can run in real-time, we simplify the problem and approximate its solution.

To this end, we reduce the number of different triangle footprints for packing. First, we eliminate rotation. Any tiny movement on the client (translation and rotation) easily moves relative sample locations by half a pixel. Similarly, aligning triangles with pixels in the atlas or rotating them changes relative sample locations. Thus, while it is desirable to match the target sampling pattern as close as possible, rotation or movement in the atlas has little effect on the final sampling pattern. Thus, we eliminate rotation as a variable and rotate all triangles in such a way that the longest edge follows the x-axis, reducing the different shapes of triangles for packing significantly.

From the rotated triangles, we bin them according to the angles adjoining the longest edge (α and β) and the pixel height h of the resulting triangle, as seen in Figure 4a. As mirroring a triangle along the y-axis does not change its shape, we can constrain $\alpha \geq \beta$ in all cases, see Figure 4b. Note that we do not consider the length of the longest edge for binning, as α, β and h uniquely identify the pixel length of the longest edge. However, any three parameters describing a triangle would work. Using α, β and h comes naturally for our approach, as we combine binned triangles to form snake-like stripes with a zigzag pattern. To this end, we flip every second triangle horizontally and vertically to form a stripe of height h. Thus, a pair of triangles forms one block within the bin, and neighboring blocks overlap, as shown in Figure 5.

If binning would only combine triangles of exactly the same shape, they could be combined to result in zero wasted space in between triangles and multiple stripes could be stacked on top of another to fully fill up a texture. Obviously, we need to perform actual binning of *similar* triangles as finding triangles of *exactly* same size is rather unlikely. To this end, we quantize α, β and h and map all triangles within a bin to the maximum triangle size supported

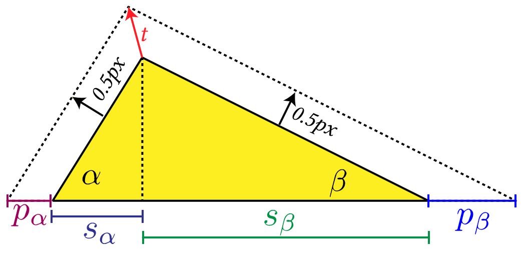


Figure 6: *Lengths s_α and s_β split the longest edge at the projection of the tip vertex. Padding lengths p_α and p_β due to half-pixel enlargement are needed to support correct bilinear interpolation when decoding. Note the tip offset t (red), which is obtained by intersection of the offset edges.*

by the bin. All triangles smaller than the biggest triangle fitting into the bin will create empty space between triangles. Obviously, the more similar triangle footprints are captured in a bin, the less space is wasted in between of triangles.

Additionally, a border between triangles is needed to support linear interpolation on the client. Thus, we place the lower triangle edge exactly along the pixel centers and the tip of triangle is only allowed to reach to the pixel center of the top most pixel within h. For the side edges, we need to insert padding. The amount of padding necessary is dependent on α and β and is determined by moving the triangle edges half a pixel in the normal direction—see Figure 6—splitting the longest edge into lengths s_α and s_β and introducing padding lengths p_α and p_β. The length of one block depicted in Figure 5 is thus $2(s_\alpha + p_\alpha + p_\beta) + s_\beta$. Note that even though enabling conservative rasterization would enlarge the triangle footprint when rasterizing into the shading atlas, we would still miss samples needed for bilinear interpolation. The more slanted the triangle footprint, the more samples would be missed despite the conservative rasterization being enabled. This effect was already described by [HSS19b] and targeted by their oversampling strategy. The samples needed for bilinear interpolation—yet missed by conservative rasterization—are depicted as green samples in Figure 9 of [HSS19b].

The edge computation is performed in atlas space and can be computed in the geometry shader. We offset each edge along its normal and find the intersections of offset 2D lines as depicted in Figure 6. We consider these steps elementary and thus omit their mathematical description.

3.3. Temporal Coherent Binning

To support real-time updates of the texture atlas, we need to be able to update the triangle-to-bin mapping. As rendering happens on the GPU it is intuitive to also perform the binning on the GPU. Thus, the binning needs to work ultimately in parallel. Typically one would use atomic operations to assign triangles to bins. However this would change the order of triangles in each bin completely every time binning is executed, even if the camera is still, much like in [HSS19b]. Obviously, this destroys the temporal coherency when encoding the atlas as a video frame and thus would strongly increase bandwidth.

We want to ensure that the order of triangles in each bin stays consistent over time. Interestingly, the GPU provides this feature during rendering through what is known as primitive order, *i.e.*, fragments must be blended in the order their generating primitives were submitted for rendering. To exploit that functionality and implement the entire binning and footprint computation in parallel, we render all primitives that should be shaded (the PVS geometry) once. We setup a virtual framebuffer, where every pixel corresponds to one specific bin. In the geometry shader, we determine the triangle footprint and bin according to α, β and h and modify the triangle such that it exactly covers the pixel associated with its bin. During fragment shading, we use the fragment-shader interlock extension, which is supported on all current graphics cards, to employ a critical section. This critical section is executed in primitive order and we simply perform an atomic add operation, yielding a unique spot in the bin, following the primitive order.

By abusing this specific feature of real-time rendering, we end up with a temporally stable order of primititves in each bin. Additionally, all computations on the primitives themselves, *i.e.*, footprint and size computations in the geometry shader, are carried out in parallel. Also the computations on different bins in the fragment shader can run completely in parallel. Only the atomic operations on the same bin must be serialized by the GPU. Even in the worst case scenario, where all triangles end up in the same bin, the geometry shader computations are carried out in parallel and can be load balanced with the ordering operations. In our experiments on desktop GPUs, we experienced hardly a performance change when enabling/disabling the critical section in the fragment shader. Furthermore the entire process fits perfectly into the traditional rendering pipeline—vertex shaders are only executed once, even when vertices are shared. Additional features such as tessellation can be used and the order of primitives generated during tessellation will also follow a consistent order. Finally, mesh-shaders could even be used and only need to be extended to integrate the binning as a final step before emitting primitives.

3.4. Bin mapping

Sampling the bin space too scarcely results in triangles of dissimilar footprints occupying the same bin, thus wasting atlas space. However, sampling the bin space densely yields many bins with very low occupancy, resulting in short bins. This is not ideal, as packing the triangles into bin strips depicted in Figure 5 increases in efficiency the more triangles end up in a bin, as the blocks overlap and the first and the last triangle in a row produce most wasted atlas space. Thus, either upon atlas reset—or periodically—we perform a bin mapping step, which maps the bins with small occupancy to the nearest bin that is full enough. Our bin mapping step is a multi-stage parallel algorithm executed as a GPU compute stage. Firstly, we categorize the bins into *Sinks*, bins with more than M triangles, *Sources*, bins with $\leq M$ triangles. The *Sink* bins are usually small in number. Secondly, each *Sink* traverses the bin space within a range R and uses atomics to update the position of every *Source* it encounters. Thirdly, each *Source* is mapped to the closest *Sink*. If there are *Sources* that were not reached by any *Sink*, they are promoted to *Sinks* even though they have less than M triangles. Figure 7 shows the atlas before and after bin mapping. Algorithm 1 depicts the bin mapping step.

Algorithm 1: Bin Mapping

```
$Sinks, Sources \leftarrow$ categorize ($allBins$, $M$);
for $P \in Sinks$ do
    $sourcesInRange \leftarrow$ getBinsInRange ($P$, $R$, $Sources$);
    for $Q \in sourcesInRange$ do
        if $Q.m = \emptyset$ or dist $(P, Q) <$ dist $(Q.m, Q)$ then
            $Q.m \leftarrow P$;
    end
end
for $P \in Sources$ do
    if $P.m = \emptyset$ then
        promoteToSink ($P$)
end
```

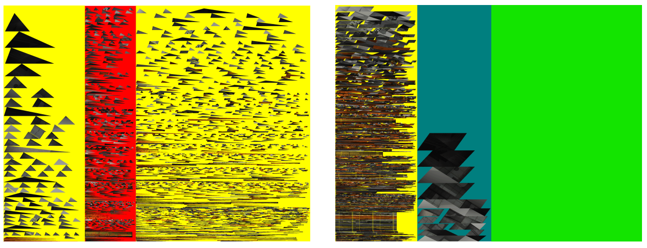

(a) *before bin mapping* **(b)** *after bin mapping*

Figure 7: *The bin distribution in superblocks before and after bin mapping. Note that the bin mapping frees up more than 50% of the atlas space.*

The values of M, R, the time interval of bin mapping and the number of overall bins are obviously related. In our setup we typically use about a million bins (1500 for h and 30 for each α and β) and $M = 16$. However, results hardly vary for $M = 8$ or $M = 32$. The range R spans 10×10 bins in the angular ($\alpha\beta$) domain and 200 in the h domain. Re-mapping the bin to a different one distorts its triangles, which potentially leads to undersampling. This is most severe when bins map to bins with smaller h. By allowing the bins to remap only upwards in the h dimension we greatly alleviate this problem.

3.5. Superblock management

As already mentioned, finding the ideal mapping of bins to a rectangular texture atlas is an NP-hard problem. We propose simplifications of the problem by packing the bins into superblocks as depicted in Figure 8. Superblocks are defined as a rectangular area of the atlas texture that contains bins. Each superblock tracks its position in the atlas, its size and the occupancy—*i.e.* the number and length of bins mapped onto the superblock. The bins are folded into rows in the superblocks, whereas superblocks have a fixed height (potentially the entire atlas) and are allocated with dynamic width. Superblocks are allocated using a simple counter and placed next to another in memory.

Superblock assignment For each bin that has not been assigned to a superblock we iterate over existing superblocks and look for the next one which has enough free space to accommodate the

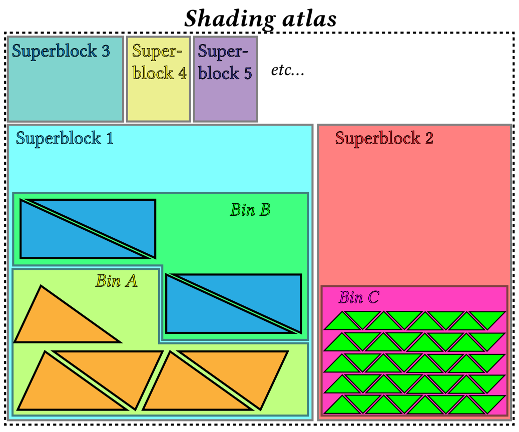


Figure 8: *An example of triangle, bin and superblock hierarchy in our shading atlas. Orange triangles belong to Bin A, blue to Bin B and green to Bin C. Bins A and B belong to Superblock 1, Bin C to Superblock 2, Superblocks 3-5 are empty.*

bin. Since the number of our superblocks usually stays under 128, their data footprint is small and thus iterating over them in compute kernels showed good performance. After being assigned to the bin, the superblock updates its occupancy info.

Superblock creation If there aren't any superblocks, or the existing superblocks are all full, we create a new one. We compute the width of packing the candidate bin into a square and round to the nearest block width (block = the pair of triangles within a bin, see Figure 5).

Superblock lifetime If the bins shrink or grow, we update the occupancy counter of the superblock. If the occupancy drops below a threshold T, the superblock collapses and becomes entirely empty, forcing the remaining bins to look for a new superblock. If the contained bins outgrow the allocated space in the superblock, they are removed from the superblock and assigned to a pool of bins scheduled for new superblock assignment.

Atlas reset The superblocks keep their position within the atlas and do not move unless the atlas runs out of free space for a new superblock. Then we perform an atlas reset—all bins are removed from the superblocks, the superblocks deallocate and cease to exist and new bin mapping and new superblocks are constructed for the current frame.

The superblock management stage is depicted in Algorithm 2.

3.6. Additional Temporal Consistency

We employ three strategies to increase the temporal coherency of our atlas: hysteresis, bin shrinking/growing, and bin freezing.

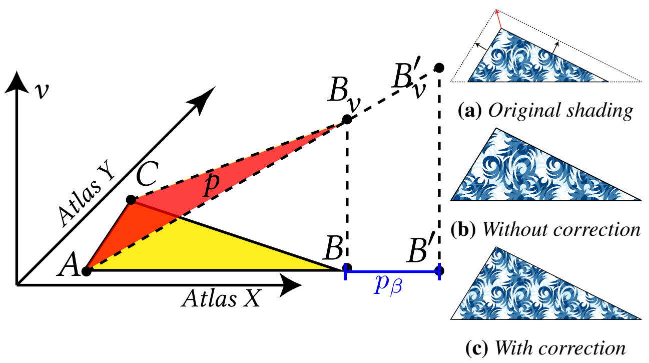
Using a hysteresis ensures triangles do not jump between bins too frequently. If a triangle did not change its footprint within a certain range, *i.e.*, it did not move too far in the bin space, we keep it in the old bin to increase temporal coherency. Thus, we allow small distortions, as the triangle is not matched with its perfect bin. For our typical setup (1500 steps in h, 30 in α, 30 in β), we use a hysteresis of 30, 5, and 5 bins in h, α, and β, respectively. These values provided sufficient balance by trading sample distribution for temporal coherency.

As mentioned in Section 3.5, bins are given a certain capacity when assigned into a superblock. Upon assignment to the superblock, the bins are assigned an extra capacity expressed as a percentage of the capacity requested. For our tests we used 30% extra buffer space. This gives the bins a space to grow and lowers the number of bin relocations needed, and thus increases temporal coherency further. If the bin shrinks below a certain percentage of its capacity (we used 50%), the capacity of the bin is shrunken and the superblock is notified that the space is no longer needed.

Finally, we freeze bin setups for a small number of frames if the camera did not move significantly. As the camera only moves little, the PVS hardly changes, which allows us to increase temporal coherency even further. In these cases (when the PVS change is below 5%), we do not allow triangles to move within bins. We free the bin setup by not clearing the previous bin setup (bin assignment, atomic counters and bin locations) and only add newly visible triangles to the bins. In this way, all previous PVS triangles stay in the atlas exactly at their previous location and new triangles are added on top. Note that hysteresis and bin freeze parameters along with view-cell size can be dynamically adjusted to respond to spikes in network connection or camera velocity.

Algorithm 2: Superblock Management

```
$orphanBins \leftarrow \emptyset$;
for $B \in allBins$ do
    if $B.superblock \neq \emptyset$ then
        $B.updateShrinkGrow(B.superblock)$;
    else
        for $S \in superblocks$ do
            if $S.freeSpace \geq B.capacity$ then
                $S.bins.insert(B)$;
                $B.superblock = S$;
                break;
        end
        if $B.superblock = \emptyset$ then
            $orphanBins.push(B)$;
end
for $S \in superblocks$ do
    if $S.occupancy < T$ then
        collapse ($S$);
        $orphanBins.push(S.bins)$;
end
if $\neg orphanBins.empty$ then $B \leftarrow orphanBins.pop()$;
while $\neg orphanBins.empty$ and $\neg atlas.isFull$ do
    $S \leftarrow$createSuperblock ($B$);
    do
        $S.bins.insert(B)$;
        $B \leftarrow orphanBins.pop()$;
    while $S.freeSpace \geq B.capacity$ and
      $\neg orphanBins.empty$
end
```



(a) *Original shading* **(b)** *Without correction* **(c)** *With correction*

Figure 9: *A depiction of 3D space used to determine correct barycentric coordinate for offset vertex* $\mathbf{B'}$. *The dimensions are atlas positions* **AtlasX**, **AtlasY** *and barycentric coordinate* **v**. *We construct a plane* **p** *from three points* $\mathbf{A} = \begin{bmatrix} x_a & y_a & 0 \end{bmatrix}$, $\mathbf{C} = \begin{bmatrix} x_c & y_c & 0 \end{bmatrix}$ *and* $\mathbf{B_v} = \begin{bmatrix} x_b & y_b & 1 \end{bmatrix}$ *(depicted in red). Querying the* **v** *coordinate of the plane at position* $\mathbf{B'} = \begin{bmatrix} x_{b'} & y_{b'} \end{bmatrix}$ *yields the correct barycentric coordinate* $\mathbf{B'_v}$ *for offset vertex* $\mathbf{B'}$. *This allows the shading to extend with the blowup while preserving the original shading within the original triangle footprint (yellow). (a) depicts the original shading and the blowup silhouette. (b) simply setting* $B'_v = 1$ *results in incorrectly streched shading. (c) computing* $B'_v = p(x_{b'}, y_{b'})$ *(and analogically for the other two vertices) results in correct shading that appears "extended".*

3.7. Shading Gathering

After the Superblock management (Section 3.5) stage we rasterize the triangles directly into the atlas using a standard forward rendering pipeline. In the geometry stage we perform a half-pixel blowup of the edges to achieve good sampling that allows bilinear interpolation when accessing the atlas, as already mentioned in Section 3.2. In the fragment stage we compute the color of the shading sample. We use barycentric coordinates $\begin{bmatrix} u & v & 1-u-v \end{bmatrix}$ to identify the shading sample location on the surface of the triangle. We assign the barycentric coordinates to the triangle vertices in the geometry stage. The built-in parameter interpolation of the standard forward rendering pipeline ensures that correctly interpolated barycentric coordinates are assigned to the fragments spawned by the triangle.

By default one would assign the default barycentric coordinates to the vertices, *i.e.* $A = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$ $B = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ and $C = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$. But due to the offset of vertex position (to do the half-pixel blowup on the triangle edges), the resulting shading would simply be stretched to the new footprint (see Figure 9b). To achieve correct shading sample distribution, we must ensure that the shading information is correct *within the original footprint of the triangle* (see Figure 9c). To this end, we alter the barycentric coordinates assigned to the vertices in the geometry processing stage in a way that "extends" the shading rather than stretching it. Let us consider triangle ABC in atlas space (Figure 9). Its footprint before the blowup is depicted as the yellow triangle. Without loss of generality, assume that the barycentric coordinates corresponding to vertex B are $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$, *i.e.* $u = 0$ and $v = 1$. The blowup of triangle edges by half pixel would offset vertex B by distance p_β to position B'.

In order to compute shading that preserves the original triangle footprint, we assign barycentric coordinates to B' in such a way that the fragment landing at position B gets the barycentric coordinates $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ assigned to it after the built-in interpolation. To compute the barycentric coordinate v at position B', we construct a plane p in 3D space x, y, v, where x and y are the atlas coordinates and v is the barycentric coordinate. This plane is defined by three points—A, C and B_v—shown in red in Figure 9. Since barycentric coordinates of vertices A and C have $v = 0$, in the x, y, v space the three points defining the plane are $A = \begin{bmatrix} x_A & y_A & 0 \end{bmatrix}$ $C = \begin{bmatrix} x_C & y_C & 0 \end{bmatrix}$ and $B = \begin{bmatrix} x_B & y_B & 1 \end{bmatrix}$. The resulting barycentric coordinate v for point B' is then obtained by querying the v value on the plane at position $x_{B'}$ and $y_{B'}$, *i.e.* $p(x_{B'}, y_{B'}) = v$. Analogically, we proceed for the blown up vertices A' and B'. We consider constructing a 3D plane from three points elementary, as well as providing two coordinates to a 3D plane and querying the third coordinate from it. Afterwards, the sample locations are adjusted for perspective foreshortening [MHAM08].

3.8. Empty Space and Encoding

All figures in this paper and frames in the supplemental video show colorful backgrounds to better depict the positioning of our superblocks. However, empty space with distinct color is sub optimal for encoding the atlas in a video stream. In practice, we use the average triangle color to clear each bin the atlas in every frame.

In order to minimize the wasted atlas space further, we fill up the empty space in bin stripes by blowing up the triangle even further; we move the edges out to match the full maximal triangle footprint a bin can accommodate. This is equivalent to mapping every triangle to the same footprint in the bin but remapping the insides to match the target triangle footprint, as mentioned in Section 3.7.

Afterwards, the shading atlas frame is encoded into a video stream with h.264 and transmitted to the client alongside the PVS geometry.

3.9. Decoding and Rendering on the Client

The client receives the PVS geometry along with the encoded shading atlas frame. The vertices of the PVS contain texture coordinates into the atlas. The sample position must be decoded with inverse perspective projection, exactly as in [HSS19b]. Thus, a correct shading sample is queried. Note that the positioning of samples in the atlas allows for correct hardware-accelerated bilinear interpolation.

4. Implementation

We implemented a prototype of the streaming rendering pipeline depicted in Figure 2, with SnakeBinning (Figure 3) implemented as a combination of CUDA compute stages and forward OpenGL rendering and C++. Of course, all stages could also be written in Vulkan or DirectX.

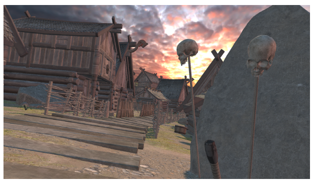
The **Assignment to Bin** stage is an OpenGL program that determines the triangle footprint in the Geometry shader and assigns the triangles to bins. Hysteresis is also performed at this step. Each bin is described by its 32-bit id, which uniquely maps to its α, β, h range. The metadata furthermore consists of corresponding superblock id (unsigned 16bit), 2D pixel coordinates within the superblock (2×16bit), triangle occupancy and maximum capacity (2×32bit). We also position the triangles in a virtual framebuffer to the fragment corresponding to the assigned bin. This is followed by a Fragment shader, where we implement the critical section using fragment shader interlock and obtain the triangle order within its assigned bin that follows the primitive order. For each triangle we track the following metadata: bin id, triangle order within bin, angles α,*beta* (4×32bit). Triangle height h is represented as with 31bit precision, and the 32nd bit tells us whether the triangle is flipped within the bin. For the perspective correction described in Section 3.7 we also need to track the view-space vertex depth as seen from the server camera. We also track the new bin id (32bit) obtained after hysteresis and/or bin-mapping.

The following stage, **Bin Mapping**, is implemented as a series of consecutive CUDA kernel executions. The first kernel categorizes all bins into *Sinks* and *Sources* (Section 3.4). Then we launch a parallel reduction kernel to construct a hierarchy in the bin space, *i.e.* an octree. Starting from each *Sink* and *Source*, we traverse the bin space neighborhood within range R. Moving up the hierarchy, *Sources* are propagated up the hierarchy. If multiple *Sources* meet in one node, the one closest to the center is chosen. If only *Sinks* meet, we propagate the *Sink* which is most central to the supported region of R, *i.e.*, the one that is closest to the center that is reached when ending at the highest level of the constructed hierarchy. If the top level of the constructed hierarchy, *i.e.*, which corresponds to a footprint of R, only contains *Sinks*, we choose the central *Sink* as *Source*.

In a second pass, we traverse the hierarchy bottom up for each *Sink*, mapping to the first *Source* stored in the hierarchy. In this way, we perform an efficient search and map to a plausible close *Source*. Note that due to the way the hierarchy is built, *Sinks* do not always find the closest *Source*, as the search is essentially limited within an R-sized grid cell. However, an exhaustive search in the complete neighborhood of each *Sink* would be too costly.

Superblock Management is also realized as a CUDA compute kernel. The metadata for each superblock consist of its pixel size (2×16bit), the height of the free spot (16bit), position within the atlas (2×16bit) and an index of the starting bin (32bit). With this info we can perform all the superblock operations from Section 3.5. We first perform the bin shrinking, thus freeing space in the superblocks, potentially collapsing superblocks that end up scarcely populated after the shrinking. The bins from deallocated superblocks join the pool of bins yet unassigned to any superblock. The bin assignment follows, potentially creating new superblocks. The bins assigned to the superblock are constructing a linked list, allowing us to efficiently traverse the bins within superblock and perform the shrink/grow operations.

The **Shading gathering** is an OpenGL rendering stage with the shading atlas texture bound to draw framebuffer. In geometry processing, we perform the edge blowup and barycentric coordinates computation and in fragment processing we correct for perspective effects of the sample locations and compute the final shading sample color and store it into the atlas.


(a) *Viking Village (4.6M triangles)*

(b) *Robot Lab (472k triangles)*

Figure 10: *Frames from the tested scenes, lit with deferred shading, PCF shadow maps, environment-map lighting and tone mapping. There are three PCF shadowmaps of* 3000×3000 *resolution in Robot Lab and one in Viking Village.*

Table 1: *We tested three PVS configurations with increasing translation buffer (size in centimeters) and rotation buffer (horizontal field of view degrees) for camera movements. The last two columns show the triangle count of the PVS, averaged over the whole walkthrough (660 frames).*

	Transl.	**FOV**	**Robot Lab**	**Viking Village**
PVS1	10 cm	90°	63.86 k	107.63 k
PVS2	20 cm	125°	82.57 k	131.30 k
PVS3	30 cm	160°	84.46 k	133.43 k

5. Evaluation

To evaluate our method, we tested two scenes also used in previous work on streaming rendering: RobotLab (RL) and VikingVillage (VV), shown in Figure 10. We at first evaluate the internals and parameters of SnakeBinning, before comparing it to Shading Atlas Streaming (SAS) [MVD*18] and Tessellated Shading Streaming (TSS) [HSS19b]. The evaluation was run on a Intel Xeon-E5 2643 CPU with 32 GB RAM and an NVIDIA TITAN RTX as server. For atlas encoding we use the nvenc encoder creating an h.264 stream. All approaches are similar in terms of client complexity—rendering the same geometry with simple bi-linear texture lookups. The client sides of all approaches run at 120+Hz on current mobile chips, such as the Oculus Quest 2. Our tests consist of gathering frames along a camera path that simulates a walk through the scene and natural interaction with the surroundings. We ran our tests for three PVS configurations for each scene described in Table 1, with increasing rotation and translation buffer (*i.e.* supported viewcell size).

5.1. Internal Workings

We tested for different sampling of the bin space. Due to our definitions, $\beta \leq \alpha \leq \gamma$, γ being the third angle in the triangle. As γ is opposite of the longest edge, it must also have the largest angle. The possible range for β is thus $0 \ldots 60$ and for α $0 \ldots 90$. Note that the smallest angle in a triangle is always ≤ 60 deg, as the sum of all angles is 180 deg and thus the extreme is reached $\alpha = \beta = \gamma = 60$ deg. Similarly, the second largest angle is bounded by 90 deg, as the extreme is reached with $\alpha = \gamma = 90$ deg, $\beta = 0$. Thus we can limit the bin ranges accordingly. However, we limit the range of α further to avoid extremely slithery triangles, with long l and small h. Such triangles pose a problem since their block width (Figure 5) tends to exceed the size of the atlas. Thus, we decided to clip the atlas footprints to a certain supported range. We use $15\,\mathrm{deg} \leq \alpha \leq 90\,\mathrm{deg}$ and $0\,\mathrm{deg} \leq \beta \leq 60\,\mathrm{deg}$.

In order to assist the bin mapping step in reducing the amount of scarcely populated bins we decided to sample the bin space with logarithmic scale. For our test scenes this step resulted in a significant improvement of the bin occupancy, as the triangle distributions are typically skewed towards smaller triangles. For example, in Robot Lab there are a few large wall and floor triangles and a very large number of small triangles which make up all the other details in the scene.

5.2. Competing Methods

Our implementation of SAS and TSS uses exactly the same shading model, PVS, camera path, server framerate, atlas size, and camera parameters as our method does *i.e.*, we only change the allocation into the atlas.

To evaluate the quality of the novel view extrapolation, we used three metrics: DSSIM [LMCB06], IW-SSIM [WL10] and FLIP [ANA*20], for which we generate a single representative value by averaging the per-pixel response. Running our tests at 100% shading rate resulted in very good DSSIM of under 0.01, but it ran up to 40% slower than TSS and SAS, and captured approximately 40% more shading samples. Both SAS and TSS focus on keeping the number of shading samples low to stay closer to the number of samples required by the original image. SAS uses a bias during the allocation to enforce this setup and TSS is constrained by the tessellation patterns for their L-packing strategy (which additionally often produces a sub-optimal sample count). To ensure a fair comparison targeting a similarly constrained atlas, we adjusted SnakeBinning to reduce the number of overall gathered samples. To this end, we multiply h with a constant factor of 0.7 before binning, reducing the memory footprint and sample count of all triangles equally. This showed little influence on image quality (increasing DSSIM by 0.04 on average) while significantly increasing speed and reducing memory requirements. Our image quality measurements can be seen in Table 3 and in Figures 11 and 12.

Our timings and sample count measurements can be seen in Table 2. The execution of our method is clearly dominated by the *Shading gathering* stage, which computes the final position of a triangle within the assigned bin and performs the blowup (Section 3.2) and barycentric coordinate computation (Section 3.7). The timings of all other stages of our SnakeBinning pipeline are summed up in the *Struct manage* entry in Table 2 and take approximately 4ms on average. Overall, we achieve the fastest server running times followed by SAS and TSS being further behind. TSS gathers shading samples by explicit execution of tessellation evaluation shaders and complex thread ID computation, which lowers their speed. SAS slightly falls behind for VV, which has smaller geometry and thus they spend more time on memory allocations compared to the raw sample counts. SnakeBinning remains mostly unaffected by triangle sizes as all stages are well parallelized and execute efficiently.

TSS achieves the fastest client rendering times. We attribute this to the fact that TSS has the smallest memory footprint of the client vertex metadata. However, all three methods show very fast client speeds, sufficient for high-resolution high-framerate rendering on thin mobile devices. Both our approach and SAS show similar timing of shading sample gathering. TSS takes nearly twice the runtime for PVS1 and PVS2 and is similar in time for PVS3.

Also, the amount of shading samples gathered is similar for SnakeBinning and SAS in Robot Lab. TSS gathers significantly more samples for PVS1 and PVS2, but drops the number of samples for PVS3. This is due to the fact that they sample the triangle under server camera projection, which has a large FOV that distorts space. This also explains the speed improvement observed for TSS PVS3, as gathering fewer samples leads to faster execution times.

Comparing the quality (Table 3) indicates that SnakeBinning is slightly better than SAS, whereas TSS is clearly worse. Inspecting example images in Figure 11 shows that SnakeBinning reduces quality uniformly with little to no artifacts that catch the attention. SAS however, shows good quality where patches in the atlas match the project size in the client view, but bad quality where this is not the case. TSS clearly shows aliasing artifacts due to missing mipmapping during shading gathering in tessellation, which does not support mipmapping.

For Viking Village—which is a more challenging and a more realistic testcase—performance is clearly different. Runtimes indicate that SnakeBinning is clearly faster than both TSS ($1.5-2.0\times$) and SAS ($2.0-3.5\times$). TSS gathers the lowest number of samples, followed by SnakeBinning and SAS. Clearly, SnakeBinning uses these samples in the best way, showing significantly better image metrics than TSS and SAS. An inspection of views in Figure 11 again confirms that SnakeBinning achieves a uniform quality and sample distribution while SAS shows spurious artifacts and overall blurry appearance due to wide FOV of the server view and block-size limitations, which were set to 256x256 in our test cases. We attribute the clear drop of SAS between RL and VV to the fact that RL has many rectangular structures which also project upright on screen and thus naturally map to the SAS atlas structure. VV on the other hand, has more natural and less regular primitives, reducing the quality with such a fixed packing strategy. SnakeBinning on the other hand, does not favor any triangle shape and thus achieves high quality throughout.

While overall all methods achieve good quality (see quality metrics above), our approach still outperforms the others as it is more consistent and avoids artifacts where other methods are limited by their predefined atlas shapes. Figure 11 shows frames where SAS and TSS artifacts are apparent.

Note that the original SAS and TSS papers report slightly different run times and quality metrics than us, which is likely due to different camera paths, shading setups, or scene preprocessing. For all our comparisons we use unaltered versions of RoboLab and VikingVillage and identical shading parameters for all methods. Preprocessing of the scene to adjust triangle sizes may reduce some of the issues for SAS [MVD*18]. However, static preprocessing cannot consider all potential views on a triangle (closeups, views with skewed angles, etc) and thus it remains questionable how preprocessing should exactly be carried out for SAS, especially as there is no public source code available.

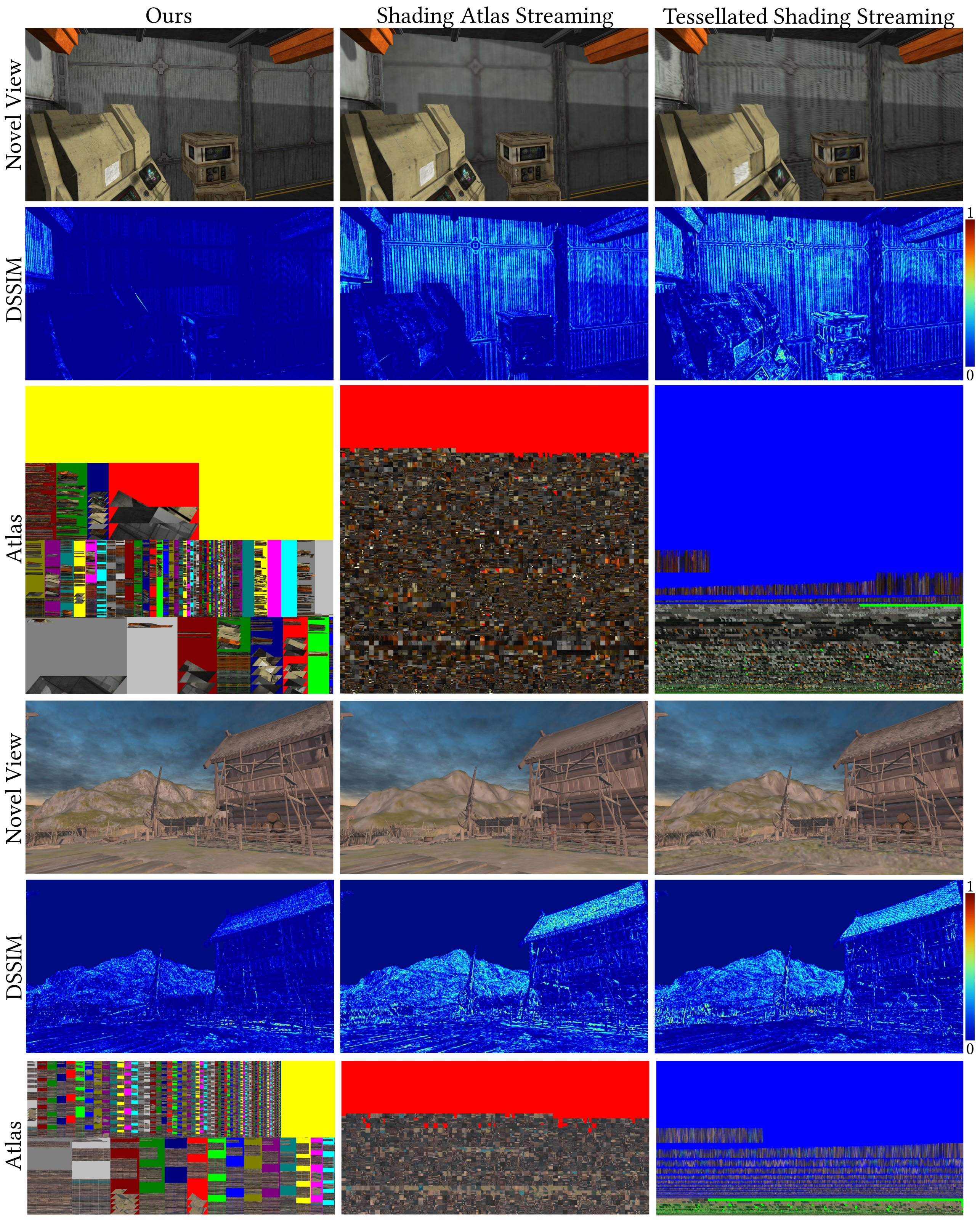


Figure 11: *Selected frames from our walkthroughs. Robot Lab uses an* 8*k* × 8*k atlas, Viking Village* 8*k* × 4*k. SAS suffers from undersampling artifacts when squares are too big to fit into the fixed block size, while TSS suffers from the lack of mip-maps in the tessellation stage. Our method outperforms them by capturing comparable amount of samples faster and distributing them more optimally.*

Table 2: *The measurements of our method, averaged over 660 frames in a walkthrough.* ***Structs*** *entails the bin assignment, mapping and superblock management stages.* ***Gathering*** *time is the atlas mapping, triangle blowup and rasterizing the shading into the atlas.* ***Server*** *is the total server time. We outperform the competition in most PVS configurations, often being faster and using fewer shading samples. Further discussion and explanation of the measurements can be found in Section 5.2.*

Scene	**SnakeBinning**					**Tessellated Shading Streaming**			**Shading Atlas Streaming**		
and PVS	Structs	Gathering	Server	Samples	Client	Server	Samples	Client	Server	Samples	Client
RL-PVS1	3.46 ms	29.37 ms	32.83 ms	8.66 M	0.23 ms	79.63 ms	23.56 M	0.18 ms	37.6 ms	7.43 M	0.52 ms
RL-PVS2	4.76 ms	38.78 ms	43.54 ms	11.68 M	0.25 ms	80.68 ms	22.43 M	0.20 ms	40.65 ms	8.21 M	0.56 ms
RL-PVS3	5.42 ms	41.69 ms	47.11 ms	12.45 M	0.26 ms	56.55 ms	13.42 M	0.17 ms	53.2 ms	13.11 M	0.50 ms
VV-PVS1	3.40 ms	6.36 ms	9.76 ms	11.48 M	0.22 ms	15.91 ms	8.68 M	0.09 ms	26.37 ms	11.39 M	0.38 ms
VV-PVS2	3.89 ms	7.59 ms	11.48 ms	14.45 M	0.22 ms	19.85 ms	10.35 M	0.15 ms	31.05 ms	19.15 M	0.37 ms
VV-PVS3	3.83 ms	7.66 ms	11.48 ms	14.66 M	0.23 ms	16.38 ms	7.04 M	0.17 ms	38.96 ms	20.64 M	0.38 ms

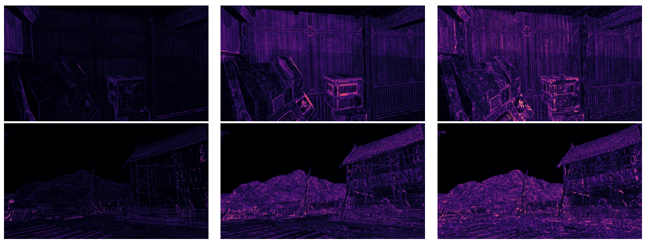


Figure 12: *The FLIP metric color mapped frames from Figure 11. Top row: Robot Lab, bottom row: Viking Village. Please zoom in for details. We used the color scale from FLIP [ANA*20].*

Table 3: *Image quality measurements using three metrics. Values are averaged over 660 frames produced in a walkthrough at 1920×1080 resolution. DSSIM and FLIP - lower is better, IW-SSIM - higher is better. SB stands for SnakeBinning - our method.*

Scene	**DSSIM**			**IW-SSIM**			**FLIP**		
and PVS	**SB**	**SAS**	**TSS**	**SB**	**SAS**	**TSS**	**SB**	**SAS**	**TSS**
RL-PVS1	.05	.04	.07	.9992	.9984	.9991	11.54	13.7	18.46
RL-PVS2	.05	.05	.08	.9983	.9979	.9982	13.82	16.74	23.96
RL-PVS3	.05	.06	.09	.9982	.9966	.9949	17.66	22.24	32.59
VV-PVS1	.06	.09	.07	.9993	.9985	.9981	8.12	10.14	15.65
VV-PVS2	.05	.09	.09	.9993	.9985	.9981	8.37	11.03	20.43
VV-PVS3	.06	.08	.10	.9993	.9957	.9908	8.97	17.99	27.76

5.3. Sample distribution

To evaluate the efficiency of the sample distribution in the atlas, we compare the texture coordinate derivatives between individual pixels during client projection to the distribution of samples in the atlas (Figure 13). Ideally, one pixel on the client screen should correspond to 1 texel in atlas space. SAS tends to undersample or oversample triangles based on how well they lend themselves to a rectangular stretch, almost never hitting the perfect 1 to 1 mapping. The fixed tessellation pattern of triangles taxes TSS; neighboring triangles often switch between oversampling and undersampling. Our method,

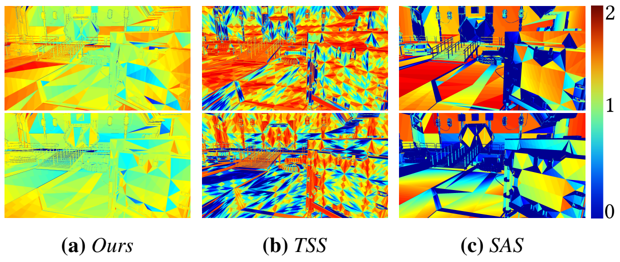


Figure 13: *The color mapped discrepancy between texture coordinate derivatives in the client view and in atlas space. Top row shows dFdx, bottom row dFdy. The camera is identical for all three methods. TSS suffers from the tessellation pattern, SAS suffers from packing all shapes into rectangles. Our strategy shows near-ideal distributions, with outliers stemming from bin mapping distortion.*

on the other hand, performs well in all cases, producing uniform sample distribution. The few outliers are due to the bin mapping, which enforces minor distortions. Note that the color mapping of Figure 13 is very strict, as a deviation of only 1 texel from the ideal is mapped to the corner colors. This shows that most of our samples exactly match the client sample distribution or deviate by less than 1 texel. Both TSS and SAS show strong temporal coherency in over- or undersampling, whereas our approach may change the sampling when bin remapping occurs. While this is strongly visible in the visualization (in the supplemental video) due to the strictness of our color mapping, the final imagery does not show any flickering.

5.4. Anisotropic Filtering and MIP Mapping

Since our method positions the shading samples on the surface of a triangle such that they match the final screen projection, all filtering considerations can already be applied during shading gathering into the atlas. This includes both mipmapping and anisotropic filtering. During generation of the client view, simple bi-linear interpolation is sufficient and we do not need to generate a texture pyramid.

While other methods do not consider perspective effects or require severe resampling for the client view generation, SnakeBinning can avoid those and achieves high image quality on the client as long as: *a*) the view stays within the target view cell and *b*) the view cell

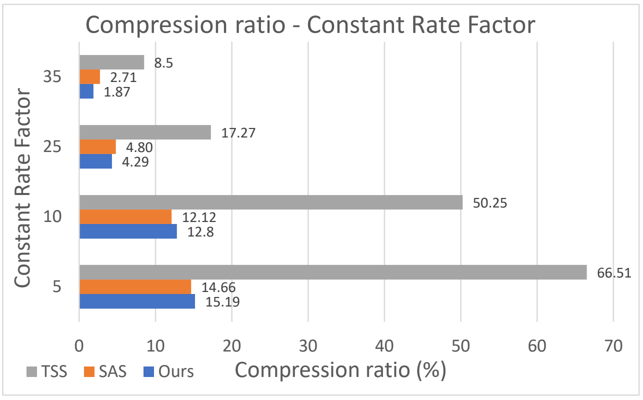


Figure 14: *The measurements of compression ratio for different values of Constant Rate Factor (CRF). Our shading atlas shows similar-or-better performance than SAS, while TSS performs poorly due to lack of temporal coherency between shading atlas frames.*

is not overly large and thus would lead to severe distortions itself. Both requirements are enforced by the streaming rendering scenario: when moving outside of the viewcell, one is missing geometry and thus severe artifacts are generated. If the viewcell is chosen too large, the amount of geometry (and shading) for transmission and update become too large.

5.5. Encoding

In order to evaluate the fitness of our approach for streaming the shading atlas over the network, we evaluate encoding bandwidth using h264. We fix the image quality via a Constant Rate Factor (CRF; a meta-parameter which roughly estimates the resulting image quality, ranging from 0 (original) to 51 (worst).) and measure the amount of compression gained over streaming just the raw shading atlas frames. The results in Figure 14 show that for CRF set to 35 and 25 we slightly outperform the compression ratio of SAS and for CRF 10 and 5 SAS slightly outperforms our method, generally staying in the same range. TSS has absolutely no temporal coherency and thus achieves the worst compression ratio. Furthermore, compressing triangular shapes generally results in worse compression ratios than squares. SAS has a clear advantage, as it distorts the triangular shapes to fit into perfect rectangles. This test shows that the temporal coherency of our atlas is on par with SAS, even though our SnakeBinning triangle strips begin and end with a sharp corner in the atlas.

Overall, our tests confirm that SnakeBinning is able to balance number of gathered samples, image quality, and temporal coherency better than any previous approach. We achieve slightly to significantly better image quality than SAS and TSS while achieving compression ratios on-par with SAS, which puts all its emphasize on temporal coherency. Thus, SnakeBinning is the preferred method for streaming a shading atlas over bandwidth limited networks for high quality rendering results.

6. Conclusion

We introduced SnakeBinning, a method for efficient capture of shading samples into a texture atlas. We use 3D binning to group triangles with similar screenspace footprint. By exploiting fragment shader interlock, we ensure the ordering of triangles in bins follows the primitive ordering, resulting in temporal coherence. Packing of bins into rectangular superblocks ensures efficient atlas space utilization. An optional bin mapping step and sampling the bin space logarithmically further improves the packing efficiency. To further improve temporal coherence, we employ a hysteresis, allocate extra space for each bin, and manage the lifecycle of superblocks by tracking the occupied space. We show that this approach gathers shading samples both ideally and temporally coherent into a shading atlas for efficient compression as a video stream for streaming rendering. By comparing to two other state-of-the-art methods for this scenario, we could clearly demonstrate the advantages of our approach in both image quality and bandwidth requirements. A parallel GPU implementations of SnakeBinning achieves real-time speeds.

While SnakeBinning seems to be close to ideal in terms of samples gathered on a triangle basis, gathering shading for other entities may still show favorable tradeoffs for the streaming rendering scenario. Looking at streaming rendering as a whole, we have not addressed the issue of view-dependent shading effects for framerate upsampling on the client.

Further research is needed in the direction of structures for representing view-dependent shading effects and their correct extrapolation, potentially incorporating these structures into the encoded atlas stream.

References

[AHTAM14] ANDERSSON M., HASSELGREN J., TOTH R., AKENINE-MÖLER T.: Adaptive texture space shading for stochastic rendering. *Computer Graphics Forum 33*, 2 (may 2014), 341–350. 3

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: FLIP: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech. 3*, 2 (2020), 15:1–15:23. URL: `https://doi.org/10.1145/3406183`, `doi:10.1145/3406183`. 10, 12

[Bak16] BAKER D.: Object space lighting. GDC, 2016. 3

[BBM*01] BUEHLER C., BOSSE M., MCMILLAN L., GORTLER S., COHEN M.: Unstructured lumigraph rendering. In *Proc. SIGGRAPH* (2001), pp. 425–432. 3

[BCC16] BOOS K., CHU D., CUERVO E.: Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *MobiSys* (2016), pp. 291–304. 3

[BCR80] BAKER B. S., COFFMAN JR E. G., RIVEST R. L.: Orthogonal packings in two dimensions. *SIAM Journal on computing 9*, 4 (1980), 846–855. 5

[BFM10] BURNS C. A., FATAHALIAN K., MARK W. R.: A lazy object-space shading architecture with decoupled sampling. In *Proc. HPG* (2010), pp. 19–28. 3

[BG04] BAO P., GOURLAY D.: Remote walkthrough over mobile networks using 3-d image warping and streaming. *IEE Proceedings - Vision, Image and Signal Processing 151*, 4 (Aug 2004), 329–336. 3

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *EGSR* (2008), pp. 1155–1164. 3

[BMS*12] BOWLES H., MITCHELL K., SUMNER R. W., MOORE J., GROSS M.: Iterative image warping. *Computer Graphics Forum 31*, 2pt1 (2012), 237–246. 3

[CG02] CHANG C.-F., GER S.-H.: *Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 1105–1111. 3

[Che15] CHEN K.: Adaptive virtual texture rendering in far cry 4. GDC, March 2015. 3

[CLM*15] CRASSIN C., LUEBKE D., MARA M., MCGUIRE M., OSTER B., SHIRLEY P., SLOAN P.-P., WYMAN C.: CloudLight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (October 2015), 1–27. 3

[COMF99] COHEN-OR D., MANN Y., FLEISHMAN S.: Deep compression for streaming texture intensive animations. In *SIGGRAPH* (1999), pp. 261–267. 3

[CTH*14] CLARBERG P., TOTH R., HASSELGREN J., NILSSON J., AKENINE-MÖLLER T.: AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Trans. on Graph. 33*, 4 (jul 2014), 1–12. 3

[CTM13] CLARBERG P., TOTH R., MUNKBERG J.: A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. on Graph. 32*, 4 (jul 2013). 3

[CW93] CHEN S. E., WILLIAMS L.: View interpolation for image synthesis. In *SIGGRAPH* (1993), pp. 279–288. 3

[CWC*15] CUERVO E., WOLMANY A., COXZ L. P., LEBECK K., RAZEENZ A., SAROIUY S., MUSUVATHI M.: Kahawai: High-quality mobile gaming using GPU offload. In *MobiSys* (May 2015), pp. 121–135. 3

[DER*10] DIDYK P., EISEMANN E., RITSCHEL T., MYSZKOWSKI K., SEIDEL H.-P.: Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *Computer Graphics Forum 29*, 2 (2010), 713–722. 3

[DRE*10] DIDYK P., RITSCHEL T., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Adaptive image-space stereo view synthesis. In *15th International Workshop on Vision, Modeling and Visualization Workshop* (Siegen, Germany, 2010), pp. 299–306. 3

[HSS19a] HLADKY J., SEIDEL H.-P., STEINBERGER M.: The camera offset space: Real-time potentially visible set computations for streaming rendering. *ACM Trans. Graph. 38*, 6 (Nov. 2019), 231:1–231:14. URL: http://doi.acm.org/10.1145/3355089.3356530, doi:10.1145/3355089.3356530. 3, 4

[HSS19b] HLADKY J., SEIDEL H.-P., STEINBERGER M.: Tessellated Shading Streaming. *Computer Graphics Forum* (2019). doi:10.1111/cgf.13780. 1, 2, 3, 5, 8, 9

[HY16] HILLESLAND K. E., YANG J. C.: Texel shading. In *Proc. Eurographics: Short Papers* (2016), pp. 73–76. 3

[KEP97] KIRULUTA A., EIZENMAN M., PASUPATHY S.: Predictive head movement tracking using a kalman filter. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 27*, 2 (1997), 326–331. 4

[LCC*15] LEE K., CHU D., CUERVO E., KOPF J., DEGTYAREV Y., GRIZAN S., WOLMAN A., FLINN J.: Outatime - using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. Mobile Systems, Applications, and Services* (2015). 3

[Lev95] LEVOY M.: Polygon-assisted jpeg and mpeg compression of synthetic images. In *SIGGRAPH* (1995), pp. 21–28. 3

[LMCB06] LOZA A., MIHAYLOVA L., CANAGARAJAH N., BULL D.: Structural similarity-based object tracking in video sequences. pp. 1 – 6. doi:10.1109/ICIF.2006.301574. 10

[LPRM02] LEVY B., PETITJEAN S., RAY N., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph. 21* (07 2002), 362–371. doi:10.1145/566654.566590. 3

[MCO97] MANN Y., COHEN-OR D.: Selective pixel transmission for navigating in remote virtual environments. *Computer Graphics Forum 16* (1997), C201–C206. 3

[MHAM08] MUNKBERG J., HASSELGREN J., AKENINE-MÖLLER T.: Non-uniform fractional tessellation. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2008), GH '08, Eurographics Association, pp. 41–45. 8

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3d warping. In *Proc I3D* (1997). 3

[MVD*18] MUELLER J. H., VOGLREITER P., DOKTER M., NEFF T., MAKAR M., STEINBERGER M., SCHMALSTIEG D.: Shading atlas streaming. *ACM Trans. Graph. 37*, 6 (2018), 199:1–199:16. 1, 2, 3, 4, 9, 10

[NCO03] NOIMARK Y., COHEN-OR D.: Streaming scenes to mpeg-4 video-enabled devices. *IEEE Computer Graphics and Applications 23* (01 2003), 58–64. 3

[Ocu18] OCULUSVR: Rendering to the oculus rift, 2018. Visited on March 30, 2018. 2, 3

[PHE*11] PAJAK D., HERZOG R., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Scalable remote rendering with depth and motion-flow augmented streaming. *Computer Graphics Forum 30*, 2 (2011), 415–424. 3

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for graphics pipelines. *ACM Trans. on Graph. 30*, 3 (may 2011), 1–17. 3

[RKR*16] REINERT B., KOPF J., RITSCHEL T., CUERVO E., CHU D., SEIDEL H.-P.: Proxy-guided image-based rendering for mobile devices. *Computer Graphics Forum 35*, 7 (oct 2016), 353–362. 3

[SGHS98] SHADE J., GORTLER S., HE L.-W., SZELISKI R.: Layered depth images. In *Proc. SIGGRAPH* (1998), pp. 231–242. 3

[SH15] SHI S., HSU C.-H.: A survey of interactive remote rendering systems. *ACM Comput. Surv. 47*, 4 (May 2015). 3

[SMSW11] SHENG B., MENG W.-L., SUN H.-Q., WU E.-H.: MCGIM-based model streaming for realtime progressive rendering. *Journal of Computer Science and Technology 26*, 1 (jan 2011), 166–175. 3

[SNC12] SHI S., NAHRSTEDT K., CAMPBELL R.: A real-time remote rendering system for interactive mobile graphics. *ACM Trans. Multimedia Comput. Commun. Appl. 8*, 3s (Oct. 2012), 46:1–46:20. 3

[TL01] TELER E., LISCHINSKI D.: Streaming of complex 3d scenes for remote walkthroughs. *Computer Graphics Forum 20*, 3 (2001), 17–25. 3

[WL10] WANG Z., LI Q.: Information content weighting for perceptual image quality assessment. ieee image proc. 20(5), 1185-1198. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society 20* (11 2010), 1185–98. doi:10.1109/TIP.2010.2092435. 10

[YN00] YOON I., NEUMANN U.: Web-based remote rendering with ibrac (image-based rendering acceleration and compression). *Computer Graphics Forum 19*, 3 (2000), 321–330. 3

[YTS*11] YANG L., TSE Y.-C., SANDER P. V., LAWRENCE J., NEHAB D., HOPPE H., WILKINS C. L.: Image-based bidirectional scene reprojection. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 150:1–150:10. 3

[Yuk17] YUKSEL C.: Mesh color textures. In *High Performance Graphics* (2017), HPG '17, pp. 17:1–17:11. 3